MITRE eCTF 2025

UIUC Design Document

Last updated	February 26, 2025
School	University of Illinois Urbana-Champaign
Team	SIGPwny
Advisor	Kirill Levchenko
Members	Minh Duong, Jake Mayer, Nikhil Date, Suchit Batpala, Akhil Bharanidhar, Stephen Cao, Sagnik Chakraborty, Sanay Doshi, Yotam Dubiner, Timothy Fong, Serkan Gozel, Yash Gupta, Adish Jain, Peter Karlos, Swetha Karthikeyan, Adarsh Krishnan, Ananth Madan, Mason Miao, Safwan Morshed, Julie Oh, Ben Pegg, Jupiter Peng, Phillip Raczka, Liam Ramsey, Neil Rayu, Rohan Seth, Preetesh Shah, Krishnan Shankar, Arpan Swaroop, Saipranav Venkatakrishnan

Table of Contents

1 Introduction **3**
- 1.1 Competition Overview . . . 3
  - 1.1.1 Security Requirements . . . 3
  - 1.1.2 Attack Scenarios . . . 4
- 1.2 Design Overview . . . 4
  - 1.2.1 Rust . . . 4
  - 1.2.2 Authenticated Encryption . . . 5

2 Security Design **5**
- 2.1 Cryptography . . . 5
  - 2.1.1 Primitives . . . 5
  - 2.1.2 Secrets . . . 6
- 2.2 Random Number Generation . . . 7

3 Functional Requirements **8**
- 3.1 Build System . . . 8
  - 3.1.1 Build Environment . . . 8
  - 3.1.2 Build Deployment . . . 8
  - 3.1.3 Build Decoder . . . 8
- 3.2 Encoder and Decoder . . . 9
  - 3.2.1 Generate Subscription . . . 9
  - 3.2.2 Encode Frame . . . 9
  - 3.2.3 Update Subscription . . . 10
  - 3.2.4 Decode Frame . . . 10
  - 3.2.5 List Subscriptions . . . 11
- 3.3 Flash Layout and State . . . 11

4 Security Requirements **12**
- 4.1 Security Requirement 1 . . . 12
- 4.2 Security Requirement 2 . . . 13
- 4.3 Security Requirement 3 . . . 13
- 4.4 Security Requirement 4 . . . 13

1 Introduction

This document describes UIUC's implementation of a secure satellite TV system for the MITRE eCTF 2025 competition.

1.1 Competition Overview

eCTF 2025 simulates a satellite network through the supplied host tools provided by the competition organizers. These host tools simulate the uplink, satellite, and downlink components. These components are out-of-scope for implementation and teams instead focus on implementing the Encoder and Decoder components, as well as the build system for these components.

- **Encoder**: Transforms input data streams into encoded streams, which is sent to the uplink, then to the satellite. The satellite then transmits one-way to the receiver, which passes the data to the Decoder.
- **Decoder**: Receives encoded streams from the receiver, decodes them, and outputs the original data streams.

The Encoder is meant to run on any standard operating system as its interface is required to be compatible with the host tools written in Python. The Decoder runs on Analog Devices' MAX78000FTHR development board, which features the MAX78000 microcontroller.

1.1.1 Security Requirements

Below are the Security Requirements (SRs) provided by the competition organizers, which are meant to defend against the Attack Scenarios.

- **SR1**: An attacker should not be able to decode TV frames without a Decoder that has a valid, active subscription to that channel.
- **SR2**: The Decoder should only decode valid TV frames generated by the Satellite System the Decoder was provisioned for.
- **SR3**: The Decoder should only decode frames with strictly monotonically increasing timestamps.

Due to the embedded systems focus of the competition, we believe that it is necessary to introduce an additional security requirement to defend against hardware attacks:

- **SR4**: The Decoder's operations should be resistant to side-channel analysis, fault injection, and other physical attacks.

1.1.2 Attack Scenarios

The following Attack Scenarios are considered by the competition organizers. Successful attacks against these scenarios result in a flag, which is a string in the form of `ectf{...}`.

- **AS1** - Expired Subscription: Read frames from a channel you have an expired subscription for.
- **AS2** - Pirated Subscription: Read frames from a channel you have a pirated subscription for (i.e. a subscription that was intended for another Decoder).
- **AS3** - No Subscription: Read frames from a channel you have no subscription for.
- **AS4** - Recording Playback: Read frames from a recorded channel you currently have a subscription for, but didn't at the time of the recording.
- **AS5** - Pesky Neighbor: Spoof the signal of the satellite to cause your neighbor's decoder to decode your frames instead.

1.2 Design Overview

To effectively address the Security Requirements and Attack Scenarios, we made the following design decisions.

1.2.1 Rust

Rust is a systems programming language that provides memory safety, thread safety, and other features that make it well-suited for embedded systems development. Rust's strong type system and ownership model helps prevent common programming errors that can lead to security vulnerabilities, such as buffer overflows.

We decided to implement both the Encoder and Decoder in Rust to take advantage of these features. To support the required Python interface for the Encoder, we used PyO3 to generate Python bindings for our Rust code and Maturin to build and pre-compile wheel files for local installation. The performance of our Encoder is also much improved by using Rust, as it is able to encode frames much faster than a Python implementation.

The MAX78000 microcontroller is a dual-core MCU with an Arm Cortex-M4 and a RISC-V core. We primarily use the CM4 core for our implementation, and leave the RISC-V core disabled. Arm targets are well-supported by Rust; however, the MAX78000 lacks a widely-supported Rust Hardware Abstraction Layer (HAL). We had previously worked with the MAX78000 in Rust before, so we decided to open source our own HAL implementation as the max7800x-hal crate. Our primary hope in publishing our HAL is that more teams will adopt Rust in their own designs.

1.2.2 Authenticated Encryption

We use the Ascon-128 authenticated encryption scheme to protect the confidentiality and integrity of encoded streams and channel subscription updates. Ascon-128 is a lightweight authenticated encryption scheme that is well-suited for embedded systems, along with countermeasures against side-channel attacks.

2 Security Design

2.1 Cryptography

At a high level, subscriptions are encrypted by authenticated encryption using a device-specific key. We aim to prevent retargeting or forging for a new device even in the event that one device is completely compromised. Additionally, subscriptions cannot be retargeted to a new channel even in the case of a compromised device key as every channel has a channel-specific key. On the other hand, frames are encrypted using a frame-specific key to ensure that retargeted frames are not even decrypted into their correct plaintext.

In order to prevent key reuse, we use a key derivation function (KDF) to derive unique keys for Ascon in various contexts. For our KDF, we chose the Keccak Message Authentiation Code (KMAC) construction defined in NIST SP800-185.

We chose KMAC because it is fast and is more efficient than a traditional HMAC construction. This is in contrast to other KDFs like PBKDF2 or Argon2, which are designed to be slow to account for low entropy in passwords. All of our KDF usage will be keyed by 256 bits of entropy (32 bytes) from base secrets. Additionally, KMAC is an eXtendable Output Function (XOF) which allows us to derive keys of arbitrary length if needed.

2.1.1 Primitives

$\text{Ascon128}_{\text{Enc}}(K, P, D, N) \rightarrow (C, T)$
$\text{Ascon128}_{\text{Dec}}(K, C, T, D, N) \rightarrow (P)$
Ascon-128 is an authenticated encryption scheme that takes a key K, a plaintext message P, associated data D, and a public nonce N as input. It outputs a ciphertext C and an authentication tag T. For decryption, it takes a key K, a ciphertext C, authentication tag T, associated data D, and a public nonce N as input. It outputs the plaintext message P.

$\text{KMAC}(K, X, L, S)$
KMAC is Keccak's keyed message authentication code that takes a key K, a message X, and a domain separator S as input. It outputs a tag of length L.

2.1.2 Secrets

None of the secrets are stored and compiled into the firmware. Instead, they are injected into reserved flash regions after the firmware is built. In order to use secrets, they must be read into RAM. We derive **`Zeroize`** traits on all sensitive secret types to allow us to zeroize them in memory after they are no longer needed or dropped to mitigate attacks related to memory leakage.

General Symbols

D - Decoder ID (4B)
C - Channel ID (4B)
T_{start} - Start timestamp (8B)
T_{end} - End timestamp (8B)
T_{frame} - Frame timestamp (8B)

Deployment Secrets

$S_{\text{base\_sub}}$ - Base subscription secret (32B)
$S_{\text{base\_chan}}$ - Base channel secret (32B)
K_{frame} - Frame key (16B)

Encoder/Decoder Secrets

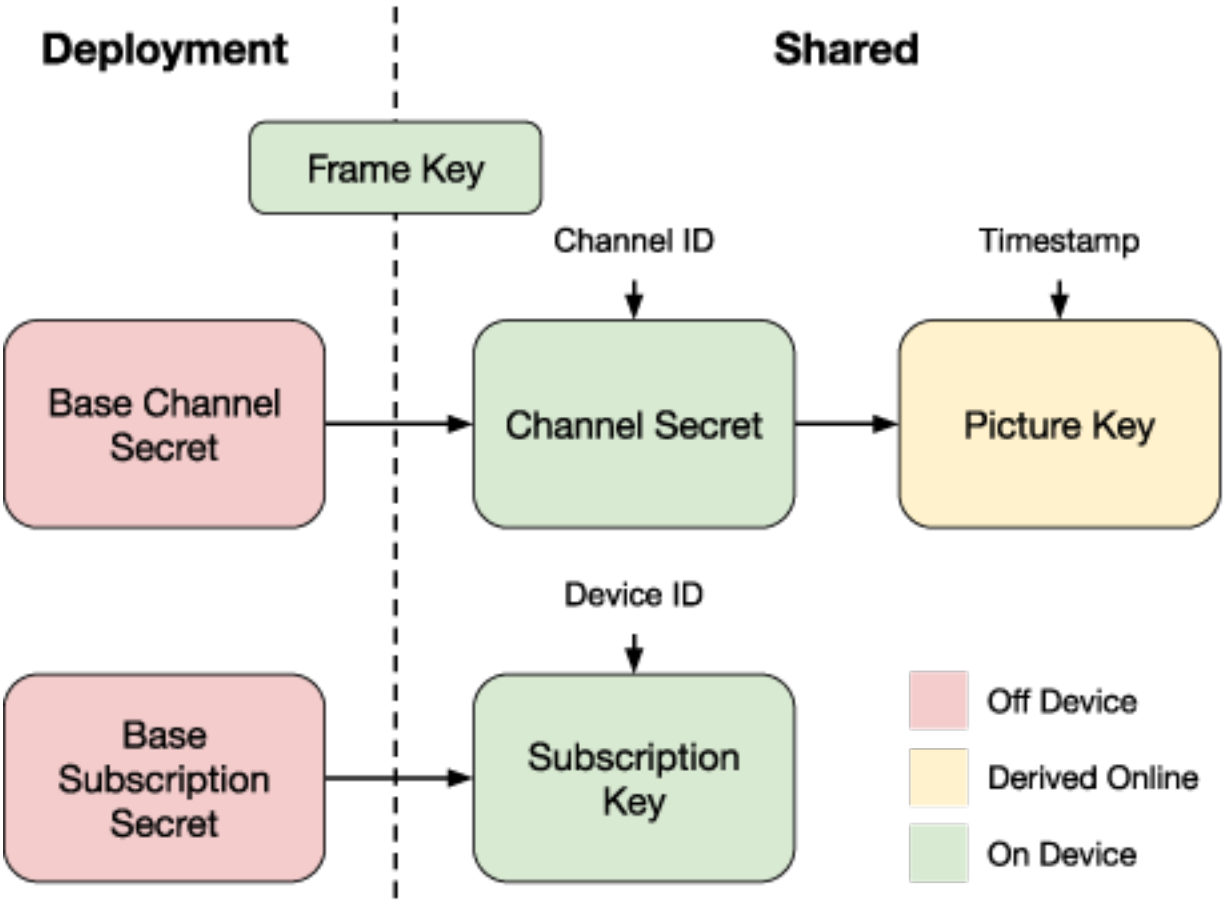
K_{sub} - Subscription key (16B), unique to decoder
S_{chan} - Channel secret (32B), unique to channel
K_{pic} - Picture key (16B), unique to channel and timestamp

Key Derivations

$$K_{\text{sub}}^{D} = \text{KMAC128}(S_{\text{base\_sub}}, D, 16\text{B}, \text{"derive\_subscription\_key"})$$

$$S_{\text{chan}}^{C} = \text{KMAC256}(S_{\text{base\_chan}}, C, 32\text{B}, \text{"derive\_channel\_secret"})$$

$$K_{\text{pic}}^{C,T} = \text{KMAC128}(S_{\text{chan}}^{C}, (T_{\text{frame}} || \sim T_{\text{frame}}), 16\text{B}, \text{"derive\_picture\_key"})$$



2.2 Random Number Generation

The MAX78000 includes a ”true” random number generator (TRNG) which is backed by hardware. However, the TRNG is a prime target for an attacker, who may attempt to manipulate or disable it. The TRNG is also very slow when it comes to sampling values.

Rather than use the TRNG directly, we opt to construct a CSPRNG upon initialization. The CSPRNG is a ChaCha20Rng construction that is seeded from a SHA3-256-HMAC construction. We first hash a static random seed from flash as an additional (but weak) source of entropy. Then, we perform 256 iterations of value collection from the TRNG, as well as the tick value from a peripheral timer which is collected immediately after the TRNG output.

We utilize the constructed CSPRNG for random delays to mitigate against fault injection attacks throughout the system.

3 Functional Requirements

3.1 Build System

3.1.1 Build Environment

We provide a Dockerfile which sets up the build environment for the Decoder. It simply installs the Rust toolchain and some **`cargo`** binaries. We have decided to use the LLVM toolchain shipped with Rust, rather than the ARM GNU toolchain to build the Decoder firmware.

- **`cargo-binutils`**: Provides quick access to LLVM tools, allowing us to run commands such as **`cargo objcopy`** or **`cargo size`** (practically drop-in replacements for **`arm-none-eabi-objcopy`** and **`arm-none-eabi-size`**).
- **`cargo-make`**: Allows running **`Makefile.toml`** files, which we use to describe the build flow for the Decoder. This is described in more detail in the "Build Decoder" section below.

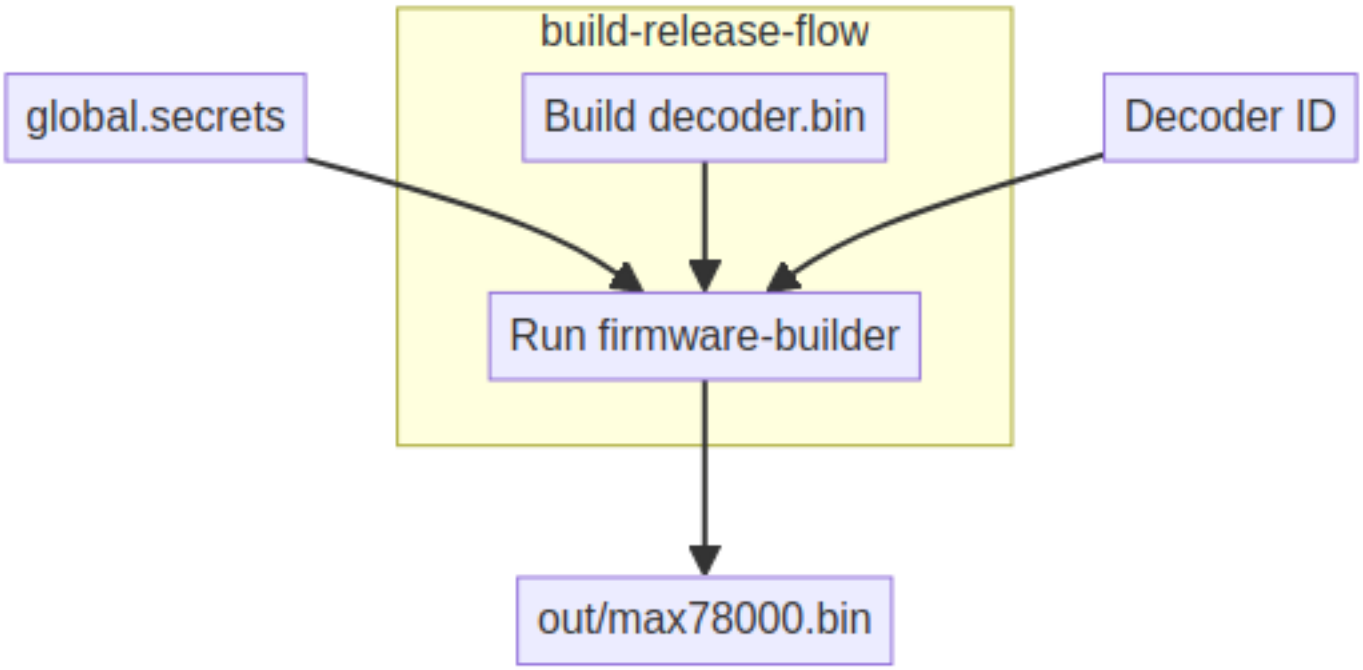
3.1.2 Build Deployment

This step is used to call **`gen_secrets`** to generate the global secrets or deployment secrets file. In our design, this step randomly generates S_{base_sub}, S_{base_chan}, and K_{frame}.

3.1.3 Build Decoder

Running the built Docker image with the expected mounts and environment variables will trigger the following build flow:

- Pre-Docker: Cleans previously copied secrets and copies the **`global.secrets`** file into the **`decoder/firmware-builder/`** directory.
- build-release-flow:
  - Builds the **`decoder.bin`** file using the **`decoder/max78000/`**.
  - Runs the **`firmware-builder`** tool to inject the secrets at the end of the firmware file according to the flash layout specification. It also injects a lifetime subscription for channel 0 to allow decoding of emergency broadcasts. This generates **`max78000.bin`**.
- Post-Docker: Copies the output **`max78000.bin`** file from the firmware-builder execution to the **`/out`** directory.



3.2 Encoder and Decoder

For most packed types/structs, we use **`bincode`** to serialize and deserialize the data.

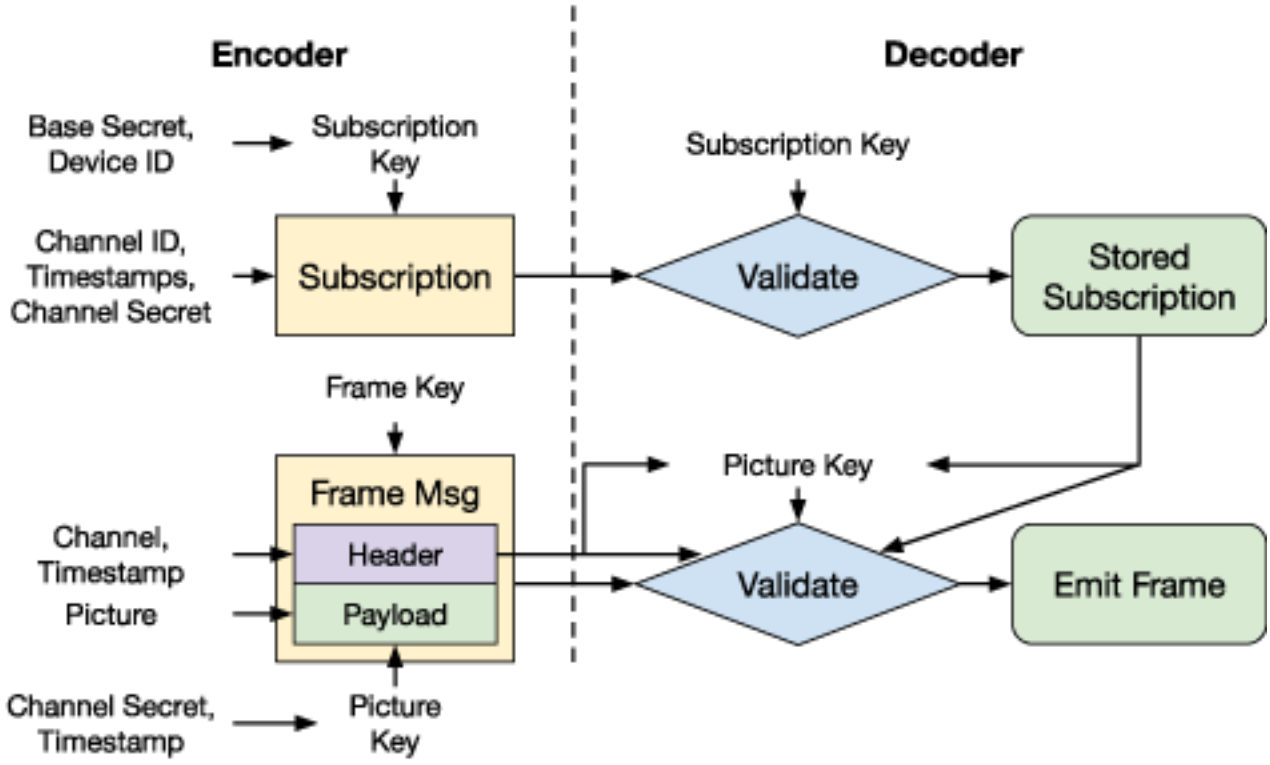
3.2.1 Generate Subscription

1. The Encoder derives the channel secret S^C_{chan} for the channel using the base channel secret $S_{\text{base\_chan}}$ and the channel ID C.
2. The Encoder creates the subscription update package by encoding the channel ID C, start timestamp T_{start}, end timestamp T_{end}, and the channel secret S^C_{chan}.
3. The Encoder derives the subscription key K^D_{sub} for the Decoder using the base subscription secret $S_{\text{base\_sub}}$ and the Decoder ID D.
4. The Encoder encrypts the subscription update package using the subscription key K^D_{sub}.
5. The Encoder emits the subscription update package to be sent to the Decoder.

3.2.2 Encode Frame

1. The Encoder derives the channel secret S^C_{chan} for the channel using the base channel secret $S_{\text{base\_chan}}$ and the channel ID C.
2. The Encoder derives the picture key $K^{C,T}_{\text{pic}}$ for the channel and timestamp using the channel secret S^C_{chan} and the frame timestamp T_{frame}.
3. The Encoder encrypts the picture using the picture key $K^{C,T}_{\text{pic}}$.
4. The Encoder packs C, T_{frame}, and the encrypted picture into a frame package.
5. The Encoder does an outer layer of encryption using frame key K_{frame}.

6. The Encoder emits the encrypted frame to be sent to the Decoder.



3.2.3 Update Subscription

1. The Decoder receives the subscription update package.
2. The Decoder retrieves its own subscription key K^D_{sub} from flash.
3. The Decoder decrypts and validates the subscription update package using K^D_{sub}.
4. The Decoder searches for a matching channel ID C in the subscription table in flash.
    - If an empty slot is found, the Decoder writes the new subscription to flash.
    - If a match is found, the Decoder erases the old subscription and writes the new subscription to flash.
    - If no match is found and no empty slot is found, the Decoder rejects the subscription update.

3.2.4 Decode Frame

1. The Decoder receives the encrypted frame.
2. The Decoder decrypts the outer layer of the frame using the frame key K_{frame}.
3. The Decoder retrieves the channel ID C and timestamp T_{frame} from the frame metadata.
4. The Decoder searches for a matching channel ID C in the subscription table in flash.

- If no match is found, the Decoder rejects the frame.

5. The Decoder verifies that the timestamp T_{frame} is between T_{start} and T_{end} of the subscription. It also verifies that the timestamp is strictly greater than the last decoded timestamp.
   - If the timestamp is not valid, the Decoder rejects the frame.
6. The Decoder derives picture key $K_{\text{pic}}^{C,T}$ for the channel and timestamp using the channel secret S_{chan}^{C} from the subscription and the frame timestamp T_{frame}.
7. The Decoder decrypts the picture using the picture key $K_{\text{pic}}^{C,T}$.
8. The Decoder displays the picture.

3.2.5 List Subscriptions

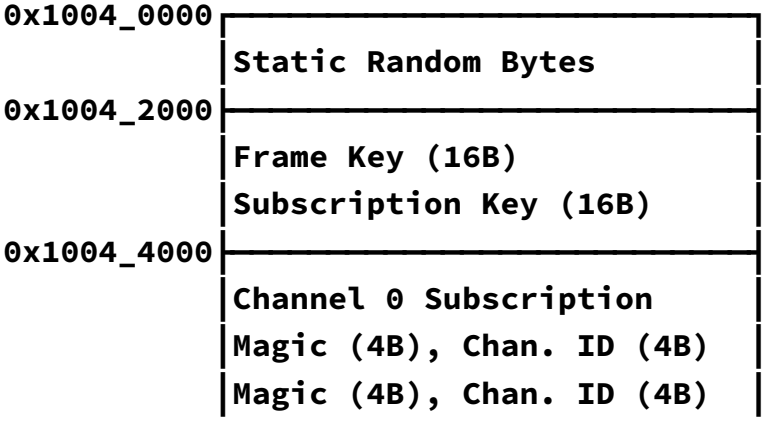
1. The Decoder iterates over the subscription table in flash and retrieves the channel ID, start timestamp, end timestamp, for each subscription.
2. The Decoder emits the list of subscriptions.

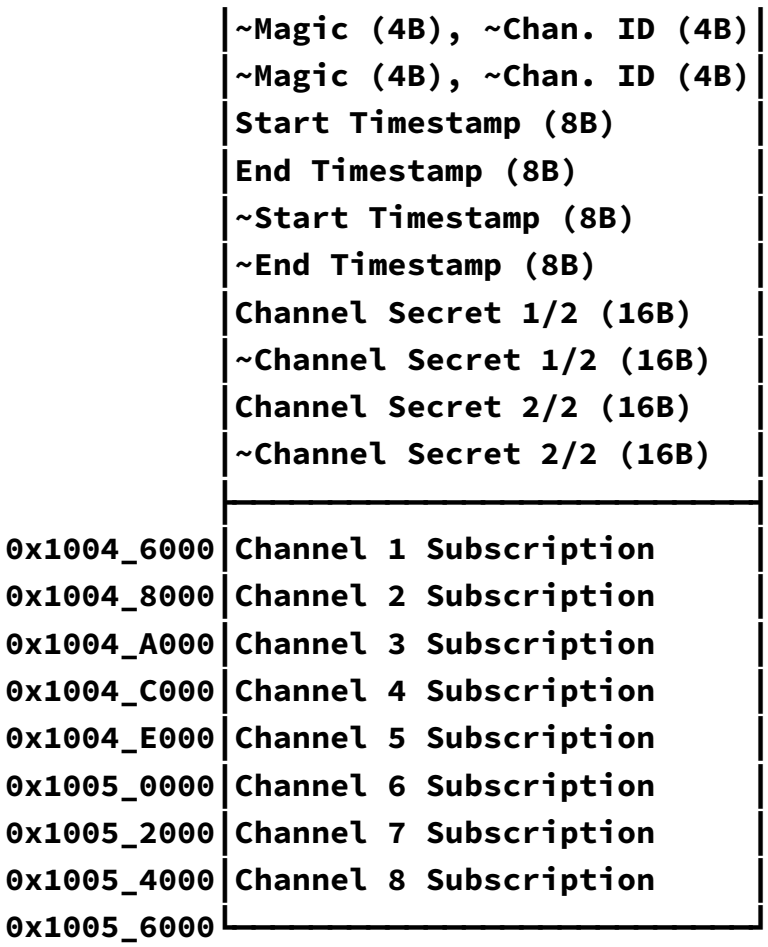
3.3 Flash Layout and State

We created a flash layout to store the Decoder's state. This includes the frame key and the subscription key. It also includes the subscription table, which can fit up to 9 entries. Each entry in the subscription table is stored in a separate flash page to allow for easy erasure and writing of subscriptions.

Note that the 0 index of the subscription table at address **`0x1004_4000`** is pre-populated with a lifetime subscription for channel 0 to allow decoding of emergency broadcasts. This is inserted by the **`firmware-builder`** tool, and also is the last page of flash that the firmware binary can occupy so that later pages can be indexed into (and are empty).

The subscription entries are stored in a packed format in little-endian order. Every 16 bytes requires the next 16 bytes to be its binary complement - this is used to detect potential flash corruption and invalidate subscription entries should this occur. The magic byte is **`0x53`**.





4 Security Requirements

Below, we summarize how we address the Security Requirements in our design.

4.1 Security Requirement 1

> ”An attacker should not be able to decode TV frames without a Decoder that has a valid, active subscription to that channel.”

By using Ascon-128 with nonces, we can validate the integrity of the picture metadata (including channel ID and timestamp) by decrypting the outer layer. By deriving a picture key using a channel secret that can only be shared via a subscription, a Decoder cannot decode a picture without a proper subscription. Confidentiality of the frame is, of course, also provided by Ascon-128.

A subscription is specific to a Decoder, as it is encrypted using a subscription key which is derived from a Decoder's ID. Only the Decoder with the matching subscription key can decrypt the subscription and obtain the channel secret. Thus, a Decoder can only decode frames for channels it has an active subscription for.

4.2 Security Requirement 2

> ”The Decoder should only decode valid TV frames generated by the Satellite System the Decoder was provisioned for.”

TV picture frames are internally encrypted using a key that is unique to the channel and timestamp and derived from a channel secret. The channel secret is unique to the channel and can only be obtained by a Decoder through a valid subscription. Since subscriptions are unique to a Decoder, a Decoder can only obtain channel secrets for channels it has an active subscription for. Thus, a Decoder can only decrypt frames for channels it has an active subscription for.

4.3 Security Requirement 3

> ”The Decoder should only decode frames with strictly monotonically increasing timestamps.”

The Decoder maintains a global timestamp counter that is only updated when a frame is successfully decoded. If a frame is received with a timestamp that is less than or equal to the current timestamp, the frame is rejected. This ensures that frames are only decoded in an increasing order.

4.4 Security Requirement 4

> ”The Decoder's operations should be resistant to side-channel analysis, fault injection, and other physical attacks.”

- We implement random delays around I/O operations to make timing and fault injection attacks more difficult, since glitches need to be precisely timed.
- The use of random nonces and derived keys also helps protect against power analysis attacks.
- We implement Zeroization to ensure that sensitive data is not left in memory after it is no longer needed.
- We implement the Arm memory protection unit (MPU) to prevent unauthorized access to memory regions, as well as to prevent code execution from data regions.